REMARKS

In the Official Action mailed on **August 13, 2004** the Examiner reviewed claims 1-6, 8-16, 18-26, and 28-30. Claim 21 was objected to because of informalities. Claims 1-2, 4, 8-10, 11-12, 14, 18-20, 21-22, 24, and 28-30 were rejected under 35 U.S.C. §102(e) as being anticipated by Kosche et al. (USPN 6,718,542, hereinafter "Kosche"). Claims 3, 13, and 23 were rejected under 35 U.S.C. §103(a) as being unpatentable over Kosche in view of Crank et al. (USPN 5,583,988, hereinafter "Crank"). Claims 5-6, 15-16, and 25-26 were rejected under 35 U.S.C. §103(a) as being unpatentable over Kosche in view of SUN Microsystems, Inc. (*C User's Guide Supplement for the Forte Developer 6 update 1 (Sun Workshop 6 update 1)*, Part No. 806-6145-10, October 2000, Revision A, XP-002242198, hereinafter "SUN").

Objections to the claims

Claim 21 was objected to because of informalities.

Applicant has amended claim 21 to correct the informalities noted by the Examiner. No new matter has been added.

Rejections under 35 U.S.C. §102(e) and 35 U.S.C. §103(a)

Independent claims 1, 11, and 21 were rejected as being anticipated by Kosche. Applicant respectfully points out that Kosche teaches a method of disambiguating memory references using aliases based upon the **assigned types of the memory references** (see Kosche, col. 7, line 41 to col. 8, line 8).

In contrast, the present invention examines **explicit type casting operations** to determine whether the explicit type-casting operations violate type-casting rules (see FIG. 2 and page 8, line 10 to page 9, line 7 of the instant application). Note that an explicit type-casting operation is specified by an explicit type-casting operator which appears in the source code of a computer program. Type-casting operators are useful because they allow programmers to selectively violate strict type rules. For more information on the explicit type-casting operator, see to attached pages (94-95 and 570) from **Applications Programming in ANSI C**, Second Edition by Johnsonbaugh and Kalin, Macmillan Publishing Company, New York, 1993.

Note that a type-casting rule is different than a typing rule because by definition type-casting operations are used to make assignments between data objects that have different types. Hence, a type-casting rule does not define a type violation, but instead specifies what kinds of type violations are permissible during a type-casting operation.

There is nothing within Kosche, either explicit or implicit, which suggests examining explicit type-casting operations to determine whether the explicit type-casting operations violate type-casting rules.

Accordingly, Applicant has amended independent claims 1, 11, and 21 to clarify that the present invention examines <u>explicit</u> type casting operations to determine whether the explicit type-casting operations violate type casting rules. These amendments find support in FIG. 2 and on page 8, line 10 to page 9, line 7 of the instant application. Applicant has amended dependent claims 2 and 12 to correct antecedent basis.

Hence, Applicant respectfully submits that independent claims 1, 11, and 21 as presently amended are in condition for allowance. Applicant also submits that claims 2-6 and 8-10, which depend upon claim 1, claims 12-16 and 18-20, which depend upon claim 11, and claims 22-26 and 28-30, which depend upon claim 21, are for the same reasons in condition for allowance and for reasons of the unique combinations recited in such claims.

CONCLUSION

It is submitted that the present application is presently in form for allowance. Such action is respectfully requested.

Respectfully submitted,

By [illegible]
Edward J. Grundler
Registration No. 47,615

Date: October 6, 2004

Edward J. Grundler
PARK, VAUGHAN & FLEMING LLP
508 Second Street, Suite 201
Davis, CA 95616-4692
Tel: (530) 759-1663
FAX: (530) 759-1665

```
            /* print the dates for one month */
            for ( day = 1; day <= days_in_month; day++ ) {
                printf( "%2d", day );
                if ( ( day + day_code ) % 7 > 0 ) /* before Sat? */
                    /* move to next day in same week */
                    printf( "   " );
                else /* skip to next line to start with Sun */
                    printf( "\n " );
            }
            /* set day_code for next month to begin */
            day_code = ( day_code + days_in_month ) % 7;
        }
    }
```

Exercise

1. Write a version of the calendar program that checks that the leap year code supplied by the user is either 0 or 1.

3.7 The Cast Operator

It is possible to convert explicitly one data type to another by using the **cast operator**. The data type to which the value of the original item is converted is written in parentheses to the left of the item. For example, if `x` is of type `int`, the value of the expression

```
( float ) x
```

is the original value of `x` converted to `float`. (The type and value of `x` are unchanged.)

Example 3.7.1. Suppose that the value of the `int` variable `hits` is the number of hits that a baseball player has made and the value of the `int` variable `at_bats` is the number of official at bats by the player. To calculate the player's batting average to several decimal places and to store the average in the `float` variable `average`, we could write

```
average = ( float ) hits / ( float ) at_bats;
```

If we simply write

```
average = hits / at_bats;
```

C discards the fractional part of the quotient

```
hits / at_bats
```

because `hits` and `at_bats` are of type `int`. Writing

```
( float ) hits / ( float ) at_bats
```

causes the system first to convert `hits` and `at_bats` to `float` and then to compute the quotient.

Exercise

1. Suppose that the variables x and y are of type `float` and z is of type `double`. Write a statement that stores the quotient x/y in z. Use casts to force the computation to be carried out with `doubles`.

3.8 getchar and putchar

As an alternative to using the format descriptor %c with `scanf` and `printf` to read and write a single character, we may use the functions `getchar` and `putchar`.

The function `getchar` reads one character from the standard input, and the function `putchar` writes one character to the standard output. If we invoke `getchar` and there is no character to read, `getchar` returns the value EOF.

Every C implementation guarantees that the value of EOF is different from the integer code of every character used by that system. This allows us to detect the end of the file without confusing the end-of-file code with the code of a character.

Example 3.8.1. The following program reads the standard input and writes it to the standard output:

```
#include <stdio.h>

main()
{
     int c;

     while ( ( c = getchar () ) != EOF )
          putchar( c );
}
```

We shall trace this program, assuming ASCII coding of characters and −1 as the value for EOF. Suppose that the standard input consists of the three characters

```
xyz
```

The first time we arrive at the `while` loop, `getchar` returns the integer code of the first character, x, in the standard input. Because the ASCII decimal code of x is 120, 120 is assigned to the variable c. (Recall that the value of the expression

```
c = getchar()
```

is the value assigned.) Thus, the expression

```
( c = getchar() ) != EOF
```

becomes

```
120 != -1
```

Because 120 is not equal to −1, the expression is true. In fact, as we previously pointed out, no character code is equal to EOF. Next, the body of the `while` loop is executed. The statement

```
putchar( c );
```

writes the character x to the standard output.

case

See switch.

Cast

The cast operator is written as

(*type*)

where *type* is a data type. When the cast operator is executed, the operand is converted to *type*. As examples, if we write

```
int i = 2;
float x;
x = ( float ) i;
```

the value of i is converted to float and copied into x.
If we write

```
struct node {
     char *data;
     struct node *link;
};
struct node *ptr;
ptr = ( struct node * ) malloc( sizeof ( struct node ) );
```

storage for one structure whose members are data and link is allocated by the library function malloc. The address of this storage is converted to the type pointer to struct node and assigned to ptr.

Comments

A C comment is delimited by /* and */.

Constants

Integer constants may be written in decimal:

```
130   45   88203
```

An integer constant that begins with 0 (zero) is an octal number:

```
0130   045
```

A sequence of digits preceded by 0X or 0x is a hexadecimal number:

```
0x90A   0Xf2
```

Either lowercase a through f or uppercase A through F is acceptable.
An integer constant may be terminated by u or U, to indicate that it is unsigned, or by l or L, to indicate that it is long. If a decimal constant is not terminated with either u, U, l, or L, it is the first of the types int, long, or unsigned long in which its value can be represented. If an octal or hexadecimal constant is not terminated with either u, U, l, or L, it is the first of the types int, unsigned int, long, or unsigned long in which its value can be represented. If a decimal, octal, or hexadecimal constant is termi-

nated with eith
in which its v
terminated wit
which its value
nated with eith
A floating-p
decimal point f
nent. The integ
digits. Either th
the decimal po
floating-point c

```
2.0   4.3
```

A floating-p
by l or L to i
terminated with
A char con
nizes the follov

Constant	
'\a'	B
'\\'	B
'\b'	B
'\f'	F
'\n'	N
'\t'	H
'\v'	V
'\r'	C
'\''	S
'\ddd'	O
'\xhh'	H

A string consta

```
"This is
```

Within a string
(double quotati

continue

When the stater

```
continue;
```

is encountered
determines whe
When the sta

```
continue;
```